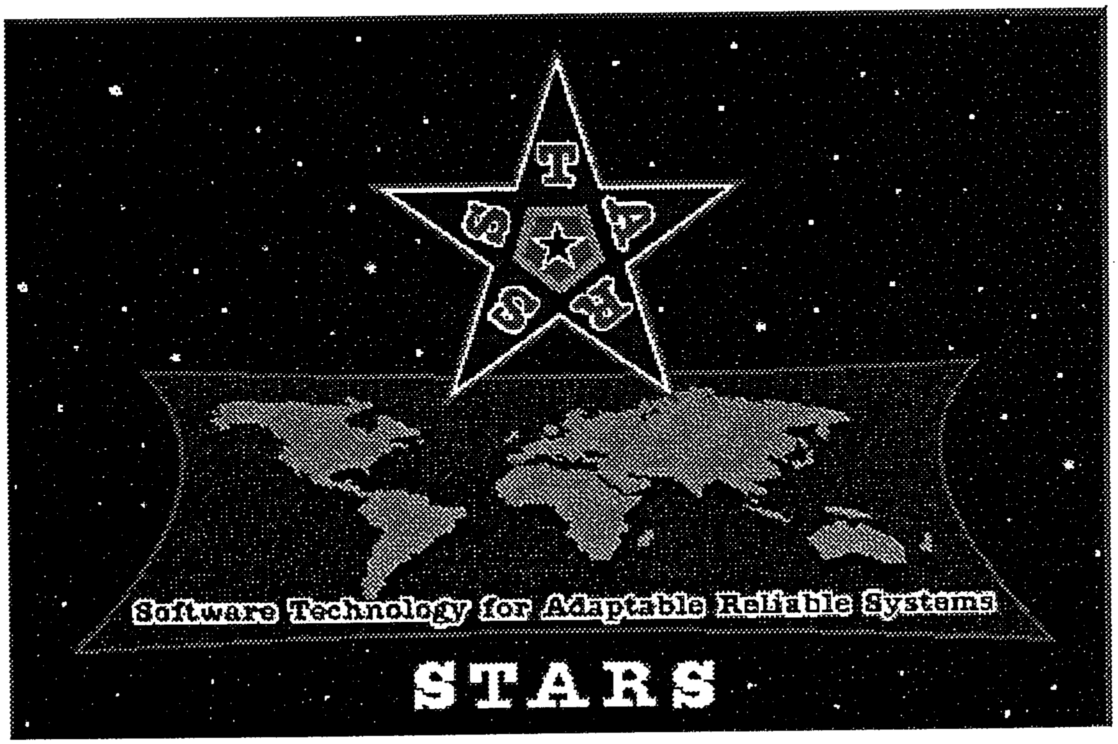
TASK: PA18
CDRL: A023
02 September 1994

Usage Report
AdaWise

Informal Technical Data



STARS-AC-A023/010/00

REPORT DOCUMENTION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	02 September 1994	Informal Technical Report

4. TITLE AND SUBTITLE

Usage Report, AdaWise

5. FUNDING NUMBERS

F19628-93-C-0130

6. AUTHOR(S)

C. A. Barbasch

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Unisys Corporation
12010 Sunrise Valley Drive
Reston, VA 22091-3499

8. PERFORMING ORGANIZATION REPORT NUMBER

CDRL NBR
STARS-AC-A023/010/00

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Department of the Air Force
ESC/ENS
Hanscom AFB, MA 01731-2816

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

A023

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution "A"

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

AdaWise is a set of tools currently under development, checks Ada programs for improper aliasing, incorrect order dependencies (including in elaboration of compilation units), and use of undefined variables. They are written in Ada, using ASIS (Ada Semantic Interface Specification) for front-end-static semantic analysis. A user of the tools must first compile the input source to be analyzed with a compiler that supports ASIS. However, while the tool set is built using ASIS, a user of the tools is *not* required to have the ASIS product itself to run the tools.

14. SUBJECT TERMS

15. NUMBER OF PAGES

13

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

TASK: PA18
CDRL: A023
02 September 1994

INFORMAL TECHNICAL REPORT

For

SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS (STARS)

Usage Report
AdaWise

STARS-AC-A023/010/00
02 September 1994

Accesion For	
NTIS CRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

CONTRACT NO. F19628-93-C-0130

Prepared for:

Electronic Systems Center
Air Force Materiel Command, USAF
Hanscom AFB, MA 01731-2816

Prepared by:

ORA
under contract to
Unisys Corporation
12010 Sunrise Valley Drive
Reston, VA 22091

TASK: PA18
CDRL: A023
02 September 1994

Data Reference: STARS-AC-A023/010/00
INFORMAL TECHNICAL REPORT
Usage Report
AdaWise

This document, developed under the Software Technology for Adaptable, Reliable Systems (STARS) program, is approved for release under Distribution "A" of the Scientific and Technical Information Program Classification Scheme (DoD Directive 5230.24) unless otherwise indicated. Sponsored by the U.S. Advanced Research Projects Agency (ARPA) under contract F19628-93-C-0130, the STARS program is supported by the military services, SEI, and MITRE, with the U.S. Air Force as the executive contracting agent. The information identified herein is subject to change. For further information, contact the authors at the following mailer address: delivery@stars.reston.paramax.com

TASK: PA18
CDRL: A023
02 September 1994

Data Reference: STARS-AC-A023/010/00
INFORMAL TECHNICAL REPORT
Usage Report
AdaWise

Principal Author(s):

____

Cheryl Barbasch *Date*

Approvals:

Teri F. Payton 9/6/94
____

Program Manager *Teri F. Payton* *Date*

(Signatures on File)

TASK: PA18
CDRL: A023
02 September 1994

Data Reference: STARS-AC-A023/010/00
INFORMAL TECHNICAL REPORT
Usage Report
AdaWise

Abstract

AdaWise, a set of tools currently under development, checks Ada programs for improper aliasing, incorrect order dependences (including in elaboration of compilation units), and use of undefined variables. They are written in Ada, using ASIS (Ada Semantic Interface Specification) for front-end static semantic analysis. A user of the tools must first compile the input source to be analyzed with a compiler that supports ASIS. However, while the tool set is built using ASIS, a user of the tools is *not* required to have the ASIS product itself to run the tools.

Contents

1 Introduction

AdaWise, a set of tools currently under development, checks Ada programs for improper aliasing, incorrect order dependences (including in elaboration of compilation units), and use of undefined variables. They are written in Ada, using ASIS (Ada Semantic Interface Specification) for front-end static semantic analysis. A user of the tools must first compile the input source to be analyzed with a compiler that supports ASIS. However, while the tool set is built using ASIS, a user of the tools is *not* required to have the ASIS product itself to run the tools.

As a preliminary test of our evolving **AdaWise** tools, we ran two of them on a variety of publicly available Ada software products. We then examined the code in those places that the tools had warned might be incorrect, to see if the code in fact contained errors. Since the tools are conservative, we were particularly interested in what percentage of the total warnings issued by the tools were actual errors. We did not attempt to find whether the tools missed any errors in the code.

We used this exercise to determine the practical value of using the tools on "real-world" code. The results of our tests indicate that the tools find actual errors, without reporting too many false warnings. We discovered in some cases that the tools issued warnings for programs that technically contained no errors, but these warnings provided useful "red flags" for programmers and future maintainers for situations in which slight modifications could cause errors later. For this reason, we envision the tools being useful during both software development and maintenance: programmers can use the tools on code as they develop it and use the tools again every time they make fixes to existing code.

The following sections give more details on our testing. We reformatted some quoted code and output from the tools to fit on the page; we did not change any of the actual text.

2 The Software Products Tested with AdaWise

To exercise the **AdaWise** tools fully and to demonstrate their applicability, we ran the tools on diverse publicly available software. We analyzed the following products.

- Arcadia's **Aflex**, a version of the flex parser in and for Ada.
- A publicly available **Dining Philosophers** program that exercises the tasking features of Ada.
- **Dhrystone**, a common benchmark of computational performance.
- Ada Standard Repository (ASR) code from SIMTEL20 (now called the Public Ada Library (PAL)):
  - **Integer Calculator**, a utility that makes infix integer calculations.

- **Line Editor**, a line-oriented file editor.
- **Expert System**, a configurable goal-driven expert system.
- **Forms Generator**, a product to create screen input forms for use in other products.
- **Menu Manager**, a product to make and use system menus.
- **Plotter**, a product that reads data points and generates video or printed output.
- **Portable Text Formatter**, the text processor and formatter used for ASR and other documents.
- **Spelling Corrector/Checker**, an interactive spelling tool with a dictionary.

Of the eleven products analyzed, four of them received warnings about potential incorrect order dependence during elaboration and three received warnings about potential improper aliasing. No one product generated more than four total warnings.

3 Elaboration Order Checking

In general, an Ada program's compilation units may be elaborated in more than one order. Chapter 10 of the Ada Reference Manual (ARM) constrains the possible orders; any order meeting these constraints is legal. If there are two different legal elaboration orders that have different observable effects, then the program has an incorrect order dependence.

The elaboration order checking tool, `check_elab`, analyzes an Ada main program and all its dependent units for incorrect order dependence in the elaboration of the compilation units. If no potential errors are reported by the tool for this program, then any legal elaboration order can be chosen by the compiler (or by another program analysis tool) without affecting execution. If the tool issues a warning, then the programmer can use `pragma ELABORATE` to eliminate the potential incorrect order dependence.

3.1 Results

We ran `check_elab` on all of the products. If a product included more than one main program, we ran the tools on each program. Four products caused warnings, and all of the warnings indicated actual incorrect order dependence in elaboration. Table 1 shows more detailed statistics.

3.2 Warnings

Table 1 shows that `check_elab` generated a total of six warnings in four products; all six warnings were for a subprogram being called from the initialization section of a package

Table 1: Statistics for `check_elab`.

Product	Source Lines	Source Units	Warnings
Aflex	11,351	50	2
Philosophers	1,093	18	2
Dhrystone	1,110	6	none
Calculator	486	7	none
Line Editor	2,646	9	none
Expert System	1,048	3	none
Forms Generator	14,198	33	none
Menu Manager	3,907	17	1
Plotter	1,094	21	none
Portable Text Formatter	10,423	34	1
Spelling Corrector/Checker	9,258	49	none

body before the subprogram's body was guaranteed to be elaborated. All of the problems would be solved by including `pragma ELABORATE` statements.

The `check_elab` tool guarantees only that there is no incorrect order dependence. If all legal elaboration orders will result in a subprogram being called before its body is elaborated, then the tool will not report the error. Once all legal elaboration orders are shown to have the same effect (no warnings are issued by `check_elab`), another tool that checks for definedness of objects can be used to check for potential raising of `PROGRAM_ERROR`.

Here is one example of the warnings issued by `check_elab` and a discussion of the potential error.

3.3 Example: Portable Text Formatter

3.3.1 Warning:

```
=> Compilation Units are NOT independent:
        dyn(A_PACKAGE_BODY)  and
        formatted_output_file(A_PACKAGE_BODY)
   (subprogram DEFINED in dyn(A_PACKAGE_BODY)
    is CALLED in formatted_output_file(A_PACKAGE_BODY) )
```

3.3.2 Explanation:

The partial order determined by context clauses does not define the order in which package body `Dyn` and package body `Formatted_Output_File` are to be elaborated. A compiler may

choose either order. The package body of `Formatted_Output_File` initializes one variable using the `D_String` function defined in package `Dyn`:

```
Header_Footer_Default
  : constant HF_LINES
    := (others => (others => Dyn.D_String(" ")));
```

This results in an unrecoverable `PROGRAM_ERROR` if (and only if) the implementation elaborates the body of `Formatted_Output_File` before elaborating the body of `Dyn`.

3.3.3 Solution:

To guarantee that no compiler generate code leaving this exception to be raised, insert `pragma ELABORATE(Dyn)` before the package body of `Formatted_ Output_File`.

4 Alias Checking in Subprogram Calls

Two program variables are aliased if their storage overlaps, so that modifying one of the variables may affect the value of the other. Unintentional or improper aliasing is a well-known source of programming errors. For example, the body of a subprogram often relies on the fact that the actual parameters matched with distinct formal parameters will not be aliased, and a subprogram call violating that assumption may behave surprisingly. These problems are compounded because, in general, the compiler may choose the order in which actual parameters are evaluated and the method by which they are passed. As a result, improper aliasing may lead not only to non-portabilities (incorrect order dependence) but also to completely undefined behavior (erroneous execution). (See the Ada Reference Manual, sections 1.6, 6.2(13), 6.4, and 12.3(17).)

The alias checking tool, `check_alias`, finds all subprogram calls and generic instantiations in a given compilation unit and checks the actual parameters (depending on mode and type) for potential aliasing with global variables, for aliasing with each other, and for independence. If no potential errors are reported by the tool, then neither the choice of parameter passing mechanism nor the order of parameter evaluation by the compiler can affect the visible behavior of the program during execution. Note that even though the tool is conservative and the warnings generated may not in fact indicate an error, the warnings can alert programmers to a potential problem that could lead to future bugs or problems in maintenance or portability.

4.1 Results

We ran `check_alias` on all of the units in each product. The tool issued warnings for three of the products. Table 2 shows more detailed statistics.

Table 2: Statistics for `check_alias`.

Product	Source Lines	Source Units	Warnings
Aflex	11,351	50	4
Philosophers	1,093	18	none
Dhrystone	1,110	6	none
Calculator	486	7	none
Line Editor	2,646	9	none
Expert System	1,048	3	none
Forms Generator	14,198	33	1
Menu Manager	3,907	17	none
Plotter	1,094	21	none
Portable Text Formatter	10,423	34	none
Spelling Corrector/Checker	9,258	19	3

4.2 Warnings

The `check_alias` tool issued a total of eight warnings in three products. This section shows those warnings and gives explanations of and solutions for each kind of warning.

4.3 Example: Aflex

`Check_alias` issued four warnings for the Aflex product: 2 warnings of potential incorrect order dependence because of aliasing of actual parameters, and 2 warnings of potential erroneous execution because of aliasing of an actual parameter with a global.

4.3.1 Warning of Incorrect Order Dependence:

```
**** dfa line  496:
DFA.EPSCLOSURE(NSET, NUMSTATES, ACCSET, NACC, HASHVAL, NSET)

=> Parameters: 6 and  1 are potential ALIASES
 (potential ORDER of COPY OUT error)
```

4.3.2 Explanation:

`Check_alias` reports that incorrect order dependence can occur. The parameters are scalar or access type, so `check_alias` does not warn of potential erroneous execution.

`DFA.EPSCLOSURE` has the following specification:

```
procedure EPSCLOSURE(T    : in out INT_PTR;
                     NS_ADDR       : in out INTEGER;
                     ACCSET        : in out INT_PTR;
                     NACC_ADDR, HV_ADDR : out INTEGER;
                     RESULT        : out INT_PTR) is
```

The reported alias involves using the same variable for the first and last (sixth) parameter. However, the last statement of the DFA.EPSCLOSURE procedure body sets the last (out) formal parameter equal to the first (in out) parameter:

```
RESULT := T;
```

Therefore, in this case, using the same variable for both parameters is not a problem since both are set to the same value on exit. Thus, the order of copy back chosen by the compiler makes no difference. This coding practice is confusing, not only to the tools but to human readers.

4.3.3 Solution:

Add a comment in DFA.EPSCLOSURE that the exit values of RESULT and T are identical; or, perhaps safer, change the code to make T mode IN or do not use aliased actual parameters.

Note that the first of these solutions does not stop the tool from issuing warning messages when you rerun the tool.

4.3.4 Warning of Erroneous Execution

```
**** main_body line  365:
EXTERNAL_FILE_MANAGER.GET_BACKTRACK_FILE(BACKTRACK_FILE)

=> Parameter  1: is aliased with a global.
 (potential ERRONEOUS EXECUTION)
```

4.3.5 Explanation:

The tool warns that an alias of BACKTRACK_FILE is updated during the call (other than by using the formal parameter).

The called procedure does not accesse BACKTRACK_FILE directly, but calls MISC.AFLEXFATAL in exception handlers, which calls MAIN_BODY.AFLEXEND directly, which uses BACKTRACK_FILE:

```
procedure GET_BACKTRACK_FILE(F : in out FILE_TYPE) is
begin
  CREATE(F, OUT_FILE, "aflex.backtrack");
exception
  when USE_ERROR | NAME_ERROR =>
    MISC.AFLEXFATAL("could not create backtrack file");
end GET_BACKTRACK_FILE;

:

-- aflexfatal - report a fatal error message and terminate
procedure AFLEXFATAL(MSG : in VSTRING) is
  use TEXT_IO;
begin
  TSTRING.PUT(STANDARD_ERROR,
    "aflex: fatal internal error " & MSG);
  TEXT_IO.NEW_LINE(STANDARD_ERROR);
  MAIN_BODY.AFLEXEND(1);
end AFLEXFATAL;
```

MAIN_BODY.AFLEXEND contains the following code:

```
if (BACKTRACK_REPORT) then
  if (NUM_BACKTRACKING = 0) then
    TEXT_IO.PUT_LINE(BACKTRACK_FILE, "No backtracking.");
  else
    if (FULLTBL) then
      INT_IO.PUT(BACKTRACK_FILE, NUM_BACKTRACKING, 0);
      TEXT_IO.PUT_LINE(BACKTRACK_FILE,
        " backtracking (non-accepting) states.");
    else
      TEXT_IO.PUT_LINE(BACKTRACK_FILE,
        "Compressed tables always backtrack.");
    end if;
  end if;
  CLOSE(BACKTRACK_FILE);
end if;
```

The predefined type TEXT_IO.FILE_TYPE is an implementation-dependent limited private type. The check_alias tool treats it as a non-scalar, without regard to a particular compiler implementation of the type. Thus, the calls to PUT and PUT_LINE potentially update BACKTRACK_FILE.

If an exception occurs attempting to create BACKTRACK_FILE, the resulting action will include attempting to write to the same file that failed, no doubt raising an unhandled exception.

This example shows a problem that is both difficult to find (it is obscured by several layers of procedure calls) and not likely to appear in testing (it happens only under error conditions), yet is potentially serious. The example also indicates that cases of aliasing tend to be trouble spots in general, and even if the situation is not "erroneous execution" in the ARM sense, it can be an actual bug.

4.3.6 Solution:

Eliminate the global alias in the call to GET_BACKTRACK_FILE, or remove the call to AFLEXEND from AFLEXFATAL.

4.4 Example: Forms Generator

Check_alias issues one warning of potential aliasing of actual parameters that could result in either incorrect order dependence or erroneous execution.

4.4.1 Warning:

```
**** form_executor lines  128 to  130:
            FORM_MANAGER.GET_FIELD_INFO
                (FIELD, NAME, POSITION, LENGTH, RENDITION,
                CHAR_LIMITS, VALUE, VALUE, MODE)

=> Parameters:  7 and  8 are potential ALIASES
 (potential ORDER of COPY OUT error) and
  (potential ERRONEOUS EXECUTION)
```

4.4.2 Explanation:

The check_alias tool warns that parameters 7 and 8 are potential aliases. The modes of the matching formal parameters are both output parameters. (If both modes were IN, check_alias would not have issued a warning.) The tool has also given an indication of the types of error caused by the aliasing. We can use the error information to inspect the code.

First, potential ORDER of COPY OUT error tells us there could be dependence on the order chosen by a compiler to copy formal variables back to actuals after execution of the body of GET_FIELD_INFO (see Ada Reference Manual, 6.4(6)).

Second, potential ERRONEOUS EXECUTION tells us that the types of both aliased parameters

are non-scalar (and non-access type) and that the effect of executing the program may depend on the parameter passing mechanism chosen by the compiler (see the Ada Reference Manual, 6.2(7)).

We inspect the code to see if there is an actual error. The procedure call listed occurs in `FORM_EXECUTOR.GET_INFO`.

The called procedure `FORM_EXECUTOR.GET_FIELD_INFO` references the abstract `FIELD_ACCESS` pointer type of the first parameter and returns some information stored in it in 8 output parameters:

```
procedure GET_FIELD_INFO (FIELD       : FIELD_ACCESS;
                          NAME        : out FIELD_NAME;
                          POSITION    : out FIELD_POSITION;
                          LENGTH      : out FIELD_LENGTH;
                          RENDITION   : out FIELD_RENDITIONS;
                          CHAR_LIMITS : out CHAR_TYPE;
                          INIT_VALUE  : out FIELD_VALUE;
                          VALUE       : out FIELD_VALUE;
                          MODE        : out FIELD_MODE) is
begin

  NAME := FIELD.NAME;
  POSITION := FIELD.POSITION;
  LENGTH := FIELD.LENGTH;
  RENDITION := FIELD.RENDITION;
  CHAR_LIMITS := FIELD.CHAR_LIMITS;
  INIT_VALUE := FIELD.INIT_VALUE;
  VALUE := FIELD.VALUE;
  MODE := FIELD.MODE;
exception
  ...
end GET_FIELD_INFO;
```

The value of `FIELD.INIT_VALUE` will presumably differ from the value of `FIELD.VALUE` in some cases where `FORM_EXECUTOR.GET_INFO` is called. This implementation depends on the returned `VALUE` being set to the `FIELD.VALUE` field. Since the parameters are non-scalar, two compiler implementations could produce different results, depending on the parameter-passing mechanism chosen, or the order chosen to copy the formals back to the actuals.

In this case, inspection shows that `check_alias` has discovered an actual error.

4.4.3 Solution:

If in fact the value of the FIELD.INIT_VALUE field is unwanted, one solution is to use a "scratch" variable to receive its value, to prevent corrupting the crucial VALUE variable.

4.5 Example: Spelling Corrector/Checker

For the Spelling Corrector/Checker product, check_alias generated three warnings: one of aliasing of two actual parameters causing potential incorrect order dependence and two of aliasing with a global causing potential erroneous execution.

4.5.1 Warning for HELP_UTILITY

```
**** help_utility.print_topic_text lines  36 to  37:
HELP_INFO_SUPPORT.APPEND_TO_DISPLAY(CURRENT_LINE.TEXT_LINE,
                                    CURRENT_LINE.LINE_LENGTH)

=> Parameter  1: is aliased with a global.
 (potential ERRONEOUS EXECUTION)
```

4.5.2 Explanation:

The potential ERRONEOUS EXECUTION warning tells us to look in the body of APPEND_TO_DISPLAY to see if the parameter passing mechanism makes a difference in the results of the execution.

HELP_UTILITY.PRINT_TOPIC_TEXT is a separately defined compilation unit:

```
separate (HELP_UTILITY)
procedure PRINT_TOPIC_TEXT (NODE: in HELP_UTILITY.HELP_LINK) is

  CURRENT_LINE: HELP_INFO_SUPPORT.TEXT_LINK;

begin

  CURRENT_LINE := NODE.TEXT_LINES;

  while CURRENT_LINE /= null loop
    HELP_INFO_SUPPORT.
      APPEND_TO_DISPLAY(CURRENT_LINE.TEXT_LINE,
                        CURRENT_LINE.LINE_LENGTH);
  CURRENT_LINE := CURRENT_LINE.NEXT_LINE;
end loop;
```

```
exception
    when others => raise;
end PRINT_TOPIC_TEXT;
```

At first glance, CURRENT_LINE appears to be a locally declared variable and thus could not be aliased with a global. But, on inspection, we see that the type of HELP_INFO_SUPPORT.TEXT_LINK is access TEXT_LINE. This means that the local CURRENT_LINE.TEXT_LINE is an object on the heap (i.e., equivalent to CURRENT_LINE.all.TEXT_LINE). Thus, check_alias considers the actual parameter to be potentially aliased with anything else of the same type on the heap.

The procedure body of HELP_INFO_SUPPORT.APPEND_TO_DISPLAY does in fact reference a global object of the same type (TEXT_LINE) on the heap:

```
procedure APPEND_TO_DISPLAY(LINE : in STRING;
                            CHAR_COUNT: in natural) is

begin

  if LINE'length <= FILE_TEXT_LINE'length then
    CURRENT_LINE.TEXT_LINE := FILE_TEXT_LINE'(others => ' ');
    CURRENT_LINE.TEXT_LINE(1..LINE'length) := LINE;
    CURRENT_LINE.LINE_LENGTH := CHAR_COUNT;
  else
    CURRENT_LINE.TEXT_LINE := LINE(1..FILE_TEXT_LINE'length);
    CURRENT_LINE.LINE_LENGTH := FILE_TEXT_LINE'length;
  end if;

  PREVIOUS_LINE.NEXT_LINE := CURRENT_LINE;
  CURRENT_LINE.NEXT_LINE := null;
  PREVIOUS_LINE := CURRENT_LINE;
  CURRENT_LINE := new TEXT_LINE;

exception
   when others =>
      raise;
end APPEND_TO_DISPLAY;
```

The CURRENT_LINE in HELP_INFO_SUPPORT is different from the CURRENT_LINE in PRINT_TOPIC_TEXT. They are not aliased, but the objects they point to (and that are referenced) are potentially aliased.

However, inspection of the body of APPEND_TO_DISPLAY shows that it never references the formal IN parameter, LINE, after updating the potential global alias. That means that even

if the parameter were aliased with a global, (i.e., the access variables had the same value), the body indeed does not have an erroneous execution because the undefined value is not subsequently used (see section 6.2(13 of the Ada Reference Manual).

This example is a common false warning, since `check_alias` does not know the values of the access variables.

4.5.3 Solution:

In this case, the tool is being overly conservative. Note the warning, perhaps leaving a comment about it, but there is no problem.

4.5.4 Warning for UTILITIES.MERGE

```
**** speller line  483:
UTILITIES.MERGE (INPUT_FILE_A, INPUT_FILE_B, INPUT_FILE_A)

=> Parameters: 1 and  3 are potential ALIASES
 (potential ORDER of COPY OUT error) and
  (potential ERRONEOUS EXECUTION)
```

4.5.5 Explanation:

The called procedure `UTILITIES.MERGE` has the following specification:

```
----------------------------------------------------------------
-- Algorithm  : This process will merge DICTIONARY_A and
--            : DICTIONARY_B into DICTIONARY_C.
----------------------------------------------------------------

    procedure MERGE (DICTIONARY_A,
                     DICTIONARY_B,
                     DICTIONARY_C : in out TEXT_IO.FILE_TYPE) is
```

Nowhere is it stated that the output dictionary may be the same as one of the input dictionaries, as is the case when it is called above. The code reads `DICTIONARY_A` and `DICTIONARY_B` into internal storage, then using the internal storage calculates the merged output and writes the results to `DICTIONARY_C`. All three files are closed at the end of `MERGE`:

```
        TEXT_IO.CLOSE (DICTIONARY_C);
```

```
if TEXT_IO.IS_OPEN (DICTIONARY_A) then
    TEXT_IO.CLOSE (DICTIONARY_A);
end if;

if TEXT_IO.IS_OPEN (DICTIONARY_B) then
    TEXT_IO.CLOSE (DICTIONARY_B);
end if;
```

Superficially there appears to be no actual error. An obvious danger might be that another programmer working on the same code (or the same programmer later) might deem the current implementation inefficient, and rewrite the routine to write the output as it reads the input files. Adding internal comments will alert future programmers to that danger.

However, even the existing program may produce unforseen results because of the aliasing. Ada's OPEN and CLOSE are not necessarily equivalent to the operating system's open and close operations. For example, closing of the file `DICTIONARY_C` might have some unforseen effect on the subsequent closing of `DICTIONARY_A` depending on whether the parameters were passed by reference or by copy, or on the implementation's association of an external file with an internal file.

4.5.6 Solution:

Change the actual parameters to MERGE so that they are not aliased.

5 Conclusions

We expected a small number of warnings for and possibly no actual errors in the products we analyzed because the products have been in use for some time. The **AdaWise** tools in fact generated only a small number of warnings. These warnings, however, indicated that there were actual errors in some products. Some warnings were "false positives". It is impossible to estimate the percentage of false positives that the tools would give on code under initial testing. An important point, though, is that almost all of the false positives in fact indicated areas of weakness in the code. We suspect that even lengthy unguided code inspections would not have revealed most of these errors and the potential bugs.